Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación
IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 5

+ + Resolución de problemas

Recordatorio

(1) Back-Tracking:

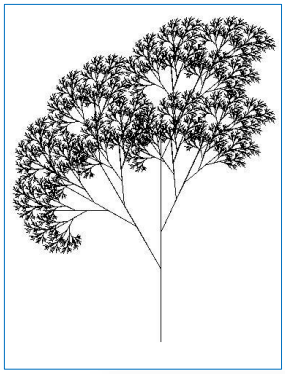
Una solución **recursiva** para problemas donde se deben **revisar exhaustivamente muchos** (a veces todos) los posibles **caminos**.

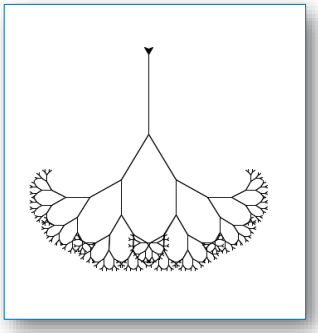
Se puede definir la solución del problema como el **recorrido exhaustivo de un "árbol invertido"**, donde cada **nodo** puede ser:

- una **bifurcación** (dos o más posibles alternativas a seguir), o
- el **término** de un camino, o
- una **solución** al problema.

Cuando se llega al término de un camino sin haber encontrado una solución, se debe **regresar** a la bifurcación previa para continuar con la siguiente alternativa. La acción de "**regreso**" es natural para la **recursión**.

(3) Ejemplos





Algoritmo de Back Tracking:
utiliza la recursión para recorrer un árbol invertido en forma ordenada

3

Backtracking

El backtracking (método de retroceso ó vuelta atrás) es una técnica general de resolución de problemas, aplicable tanto a problemas de optimización, juegos y otros tipos.

El backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar **muy costoso (en tiempo de ejecución)**.

El back-tracking es "naturalmente" recursivo.

La solución del problema se puede representar como un árbol invertido donde cada nodo representa un problema similar pero más sencillo.

Se debe recorrer cada "rama del árbol".
En una "rama" cualquiera, se termina en un "último" nodo que representa el caso base (más sencillo) de la recursión.
Si el caso base es una solución, se termina el algoritmo.
Si no lo es, se "retorna" al nodo anterior para continuar en otra rama.

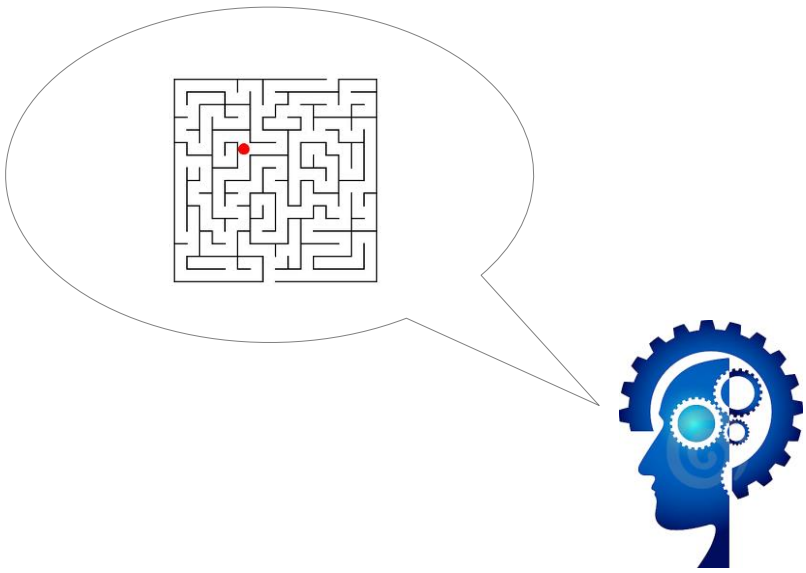
4

BACK-TRACKING:

Se deben “recorrer” muchos caminos posibles (en el “árbol invertido”) hasta encontrar la solución o decidir que no hay una solución.

Cada camino (“rama”) se recorre hasta el final: si no hay solución en esa rama, se retorna al nodo (bifurcación) anterior, para continuar con las instrucciones indicadas después del llamado a la función/método recursiva/o.

Se recorren varios caminos (“ramas”) hasta encontrar una solución (peor caso: se recorren todos los caminos posibles).



LABERINTO PROPUESTO:

```
X X X X X X X X X X X X X X X X X X X X X X
      X   X   X   X       X               X
X X   X       X       X   X   X     X   X
X         X   X X X   X   X   X X   X   X
X   X X X X   X   X   X   X X X     X   X
X       X     X   X   X             X   X
X   X X X X X X   X   X X   X               X
X                         X   X   X X   X X
X X X X X X X X X X X     X   X     X   X X
X                   X       X X X X X X   X X
X   X X X X X X       X X X   X           X
X     X           X     X           X X X
X X X X X X X X X X X X X X X X X X X X X X
```

LABERINTO PROPUESTO:

```
X X X X X X X X X X X X X X X X X X X X X X
      X   X   X   X       X               X
X X   X       X       X   X   X     X   X
X         X   X X X   X   X   X X   X   X
X   X X X X   X   X   X   X X X     X   X
X       X     X   X   X             X   X
X   X X X X X X   X   X X   X               X
X                         X   X   X X   X X
X X X X X X X X X X X     X   X     X   X X
X               X       X X X X X X   X X
X   X X X X X X       X X X   X           X
X     X           X     X           X X X
X X X X X X X X X X X X X X X X X X X X X X
```

LABERINTO RESUELTO:

```
X X X X X X X X X X X X X X X X X X X X X X
* * * X   X   X   X * * * X       * * * X
X X * X       X     * X * X   X     * X * X
X * *     X   X X X * X * X   X X * X * X
X * X X X X   X   X * X * X X X * * X * X
X * *     X     X   X * X * * * * * X X * X
X * X X X X X X   X * X * X   X       * * X
X * * * * * * * * * *   X   X   X X * X X
X X X X X X X X X X X   X   X     X * X X
X                   X       X X X X X * X X
X   X X X X X       X X X   X       * * X
X     X           X     X           X X X * *
X X X X X X X X X X X X X X X X X X X X X X
```

Diseño del Programa Python

- Supondremos que tenemos el laberinto inicial en un archivo de tipo **.csv** generado desde Excel. Le preguntaremos al usuario por el nombre del archivo y lo leeremos desde nuestro programa.
  Una **"X"** representa una pared. Un **"."** representa un espacio libre.
  No conocemos las dimensiones (**filas x cols**) del laberinto.
  Al guardar desde Excel en formato **.csv**, las celdas quedan separadas por **";"**.

- Acciones del programa:
  - Pedir al usuario que ingrese el nombre del archivo con el laberinto.
  - Leer el archivo y almacenarlo en una lista-matriz de **nfilas x ncols**.
    (Igual que en un tablero, las celda van desde **(0,0)** hasta **(nfilas-1,ncols-1)**.)
  - Imprimir el laberinto inicial.
  - Invocar **método recursivo BT** para recorrer el laberinto. Este método a su vez invoca un método para validar la siguiente posición en el laberinto.
  - Imprimir el resultado.

- Definir una **clase "Laberinto"** que contenga métodos con las acciones anteriores.

- **Propuesto:** agregar una variante al programa para imprimir los caminos intermedios recorridos antes de encontrar la solución.

Algoritmo de Solución del Laberinto

- Representar los caminos de solución del laberinto como un árbol invertido. Cada nodo del árbol representa una celda **(fila i, col j)** dentro del laberinto. Si la celda contiene una **"X"** es una pared y no podemos seguir por ese camino. Si contiene un **"."** podemos seguir recorriendo en el árbol (hacia abajo o hacia la derecha).

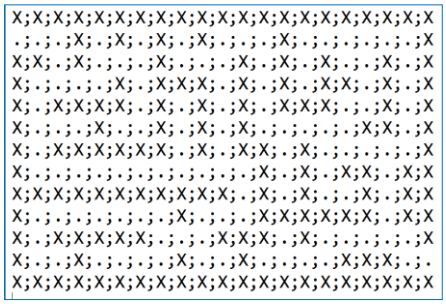
- El caso base de la recursividad (condición de término) es que las coordenadas de la celda correspondan con la salida del laberinto y se retorna **True** .

- En caso contrario, se procede a marcar la casilla y a intentar en las distintas direcciones en el laberinto (**arriba , derecha, abajo, izquierda**), asegurándose primero que en esa dirección haya una casilla válida: no es una pared, no está fuera del laberinto y no ha sido visitada anteriormente.

- En caso en que ninguna dirección nos lleve a la salida, se desmarca la casilla actual (pues se va a retroceder) y se retorna **False**.

- Las marcas sirven también para señalar la ruta que conforma la solución, una vez alcanzada.

Archivo de entrada .csv (generado desde Excel)



Podemos leer este archivo línea por línea y almacenarlo en una matriz (lista de listas).

Definimos el método **generar_lab** dentro de una **clase** "laberinto". Uno de los atributos de esta clase es **lab** y está inicializado como una lista vacía.

A cada línea le sacamos (**strip**) el símbolo **"\n"** y generamos una lista (**split**) utilizando **";"** como separador.

```
def generar_lab(self):
    arch1 = input("ingresa nombre del archivo .csv : ")
    entrada = open(arch1)
    for linea in entrada:
        linea = linea.strip("\n")
        lista = linea.split(";")
        self.lab.append(lista)
    entrada.close()
```

El nombre de archivo que se ingresa por pantalla debe incluir el tipo. Ejemplo: **Laber2in.csv**

Definición de la clase "laberinto" con sus atributos:

- Definimos como atributos/parámetros a ser definidos por el usuario cuando se invoca la creación de un objeto de **clase** "laberinto" ("**LaberPyX**" en nuestro programa), las coordenadas de inicio (**f_ini, c_ini**) y de salida (**f_end, c_end**) del laberinto.

- Dentro del método **__init__** definimos como atributos "globales" de la clase "laberinto" los posibles valores de cada celda: **marca = "*"**, **pared = "X"** y **libre = "."** También definimos en este método el atributo **lab = []** como una lista vacía a ser generada por el método **generar_lab**.

Código para definir la clase y sus atributos:

```
class LaberPyX:
    marca = "*"  # camino que recorremos
    pared = "X"  # pared del laberinto
    libre = "."  # espacio libre del laberinto

    def __init__(self,f_ini,c_ini,f_end,c_end):
        # Atributos de una instancia de laberinto:
        self.f_ini = f_ini  # fila de inicio
        self.c_ini = c_ini  # columna de inicio
        self.f_end = f_end  # fila de salida
        self.c_end = c_end  # columna de salida
        self.lab = []       #lista-matriz vacía para guardar el laberinto
        self.generar_lab() #transforma archivo .csv en lista-matriz
        self.nfilas = len(self.lab)  # número de filas en matriz lab
        self.ncols = len(self.lab[0]) #número de columnas en matriz lab

    def generar_lab(self):
        arch1 = input("ingresa nombre del archivo .csv : ")
        entrada = open(arch1)
        for linea in entrada:
            linea = linea.strip("\n")
            lista = linea.split(";")
            self.lab.append(lista)
        entrada.close()
```

Método para imprimir un laberinto:

```
    def __str__(self):
        m = ""
        for fila in range(self.nfilas):
            for col in range(self.ncols):
                m += self.lab[fila][col] + " "
            m += "\n"
        return m
```

Método para validar una celda:

```
    def valida(self,f,c):
        """ retorna True si la celda es válida y False en caso contrario """
        # revisa si la celda esta fuera del laberinto
        if (f < 0 or f >= self.nfilas) or (c < 0 or c >= self.ncols):
            return False
        # revisa si la celda ya fue visitada o es pared
        if (self.lab[f][c] == self.marca) or (self.lab[f][c] == self.pared):
            return False
        # if (lab[f][c] == libre):
        return True
```

Método para recorrer laberinto con Back-Tracking recursivo:

```
    def recorre(self,f="ini",c="ini"):
        """ método recursivo con BT que regresa cuando termina un camino
            retorna True si la celda (f,c) es la salida
            y False en caso contrario """
        # definición parámetros iniciales (comienzo del laberinto)
        if f == "ini":
            f = self.f_ini
        if c == "ini":
            c = self.c_ini
        listo = False  # aún no se encuentra la salida
        self.lab[f][c] = self.marca  # marcamos la celda como visitada
        """
        Caso Base: condición de término exitoso de la búsqueda recursiva
        """
        if (f == self.f_end) and (c == self.c_end):
            return True
        """
        Si no es el Caso Base, seguimos con algoritmo recursivo BT
        """
```

Continúa en siguiente página

Cont. método para recorrer laberinto con Back-Tracking:

```
        """
        Si no es el Caso Base, seguimos con algoritmo recursivo BT
        """
        """ (1) intentamos para arriba """
        if not listo and self.valida(f-1, c):
            listo = self.recorre(f-1, c)
        """ (2) si no está listo, intentamos para la derecha """
        if not listo and self.valida(f, c+1):
            listo = self.recorre(f, c+1)
        """ (3) si no está listo, intentamos para abajo """
        if not listo and self.valida(f+1, c):
            listo = self.recorre(f+1, c)
        """ (4) si no está listo, intentamos para izquierda """
        if not listo and self.valida(f, c-1):
            listo = self.recorre(f, c-1)
        # si llegamos al final de este camino y no tiene salida,
        # debemos desmarcarlo y retornar
        if not listo:
            self.lab[f][c] = self.libre
        # se acabó esta bifurcación del árbol de solución
        # y retornamos el resultado sea positivo o negativo
        return listo
```

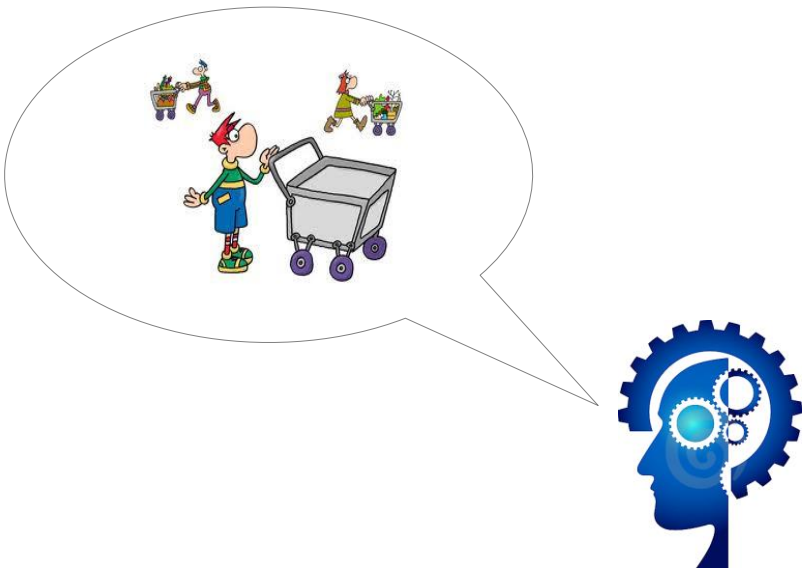
Programa ppal en siguiente página

Programa principal del Laberinto:

```
""" programa principal """
x0 = 1    # fila de inicio
y0 = 0    # columna de inicio
x1 = 11   # fila de salida
y1 = 20   # columna de salida

Lab1 = LaberPyX(x0,y0,x1,y1)
print()
print("Laberinto propuesto:\n")
print(Lab1)

if Lab1.recorre():
    print("Laberinto Resuelto!!!!")
else: print ("No tiene solución!!!")
print()
print(Lab1)
```



Problema Propuesto: Carro de supermercado

Es posible colocar los distintos productos de tal forma que solo un 10% de la capacidad del carro se pierde en los espacios entre un producto y otro.

¿Cómo determinar qué productos conviene colocar en el carro, tomando en cuenta sus volúmenes y precios asociados?

Hacer un programa que busque una configuración óptima tal que maximice el valor (precio) del contenido del carro.No interesa el detalle de los productos a elegir (pues quedan en el carro).

Asume que posee dos listas ya inicializadas, volumenes[] y precios[], con los volúmenes y precios de los productos, respectivamente (del mismo largo). Es decir, al producto **i** le corresponde **volumen [i]** y **precio [i]** . También es conocido el valor **VC** que representa el volumen total del carro.

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 5

+ + Resolución de problemas